19. The Global Alignment Module

Created: September 16, 2003
Updated: September 26, 2003

The Global Alignment Library [**`xalgoalign`**:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The library contains C++ classes encapsulating pairwise global sequence alignment algorithms frequently used in Computational Biology and applications.

- ***CNWAligner*** is a base class for global alignment algorithm classes. The class implements a generic Needleman-Wunsch algorithm producing pairwise alignment of nucleotide or protein sequences. The implementation uses affine penalty model and supports end space free mode, useful in many applications where ends of sequences may not align. The classical Needleman-Wunsch algorithm is known to have quadratic memory and CPU requirements which often seriously limits its application. However, in presence of partial alignment patterns such as ordered high scoring pairs, the problem can be split into a number of smaller ones altogether imposing less space and CPU requirements. CNWAligner utilizes this idea by providing a way to specify guides representing portions of expected or desired alignment.
- ***CMMAligner*** follows the Hirschberg's divide-and-conquer approach (developed also by Myers and Miller) under which the amount of space required to align two sequences globally becomes a linear function of sequence's lengths. While the latter is achieved at a cost of up to twice longer running time, a multithreaded version of the algorithm can run even faster than the classical Needleman-Wunsch in a multiple-CPU environment.
- ***CSplicedAligner*** is an abstract base for algorithms calculating cDna/mRna-to-genomic, or spliced alignments. Spliced alignment algorithms specifically accounts for splice signals in their dynamic programming recurrences resulting in better alignments for these particular but very important types of sequences.

Chapter Outline

The following is an outline of the chapter topics:

- Computing pairwise global sequence alignments
    - Initialization
    - Parameters of alignment
    - Computing
    - Alignment transcript
- Aligning sequences in linear space
    - The idea of the algorithm
    - Implementation
- Computing spliced sequences alignments
    - The problem
    - Implementation
- Formatting computed alignments
    - Formatter object

Demo Cases [src/algo/align/demo/nwa] [src/algo/align/demo/splign]

Computing pairwise global sequence alignments

Generic pairwise global alignment functionality is provided by CNWAligner. This functionality is discussed in the following topics:

- Initialization
- Parameters of alignment
- Computing
- Alignment transcript

Initialization

Two constructors are provided to initialize the aligner:

```
CNWAligner(const char* seq1, size_t len1,
           const char* seq2, size_t len2,
           const SNCBIPackedScoreMatrix* scoremat = 0);

CNWAligner(void);
```

The first constructor allows to specify sequences and the score matrix at the time of the object's construction. Note that the sequences must be in proper strands, as the aligners do not build reverse complimentaries. The last parameter must be a pointer to a properly initialized ***SNCBI-PackedScoreMatrix*** object or zero. If it is a valid pointer, then the sequences are verified against the alphabet contained in the ***SNCBIPackedScoreMatrix*** object and its score matrix is further used in dynamic programming recurrences. Otherwise, sequences are verified against IUPACna alphabet and match/mismatch scores are used to fill in the score matrix.

The default constructor is provided to support reuse of aligner object when many sequence pairs share same type and alignment parameters. In this case, the following two functions must be called prior to computing the first alignment in order to load the score matrix and the sequences:

```
void SetScoreMatrix(const SNCBIPackedScoreMatrix* scoremat = 0);
void SetSequences(const char* seq1, size_t len1,
                  const char* seq2, size_t len2,
                  bool verify = true);
```

where the meaning of `scoremat` is the same as above.

Parameters of alignment

CNWAligner realizes affine gap penalty model, which means that every gap of length L (with possible exception of end gaps) contributes Wg+L*Ws to the total alignment score, where Wg is a cost to open the gap and Ws is a cost to extend the gap by one basepair. These two parameters are always in effect when computing sequence alignments and can be set with

```
void SetWg  (TScore value); // set gap opening score
void SetWs  (TScore value); // set gap extension score
```

To indicate penalties, both gap opening and gap extension scores are assigned with negative values.

Many applications (such as the shotgun sequence assembly) benefit from a possibility to avoid penalizing end gaps of alignment, since the relevant sequence's ends may not be expected to align. ***CNWAligner*** supports this through a built-in end-space free variant controlled with a single function:

```
void SetEndSpaceFree(bool Left1, bool Right1, bool Left2, bool Right2);
```

The first two arguments control the left and the right ends of the first sequence. The other two control the second sequence's ends. True value means that end spaces will not be penalized. Although arbitrary combination of end-space free flags can be specified, judgement should be used in order to get plausible alignments.

The following two functions are only meaningful when aligning nucleotide sequences:

```
void SetWm  (TScore value); // set match score
void SetWms (TScore value); // set mismatch score
```

The first of them sets a bonus associated with every matching pair of nucleotides. The second function assigns a penalty for every mismatching aligned pair of nucleotides. It is important that values set with these two function will only take effect after ***SetScoreMatrix()*** is called (with a zero pointer, which is the default).

One thing that could limit a scope of global alignment applications is that the classical algorithm takes quadratic space and time to evaluate the alignment. One way to deal with it is to use the linear-space algorithm encapuslated in ***CMMAligner***. However, when some pattern of alignment is known or desired, it is worthwhile to explicitly specify "mile-posts" through which the alignment should pass. Long high-scoring pairs with 100% identity (no gaps or mismatches) are typically good candidates for them. From the algorithmic point of view, the pattern splits the dynamic programming table into smaller parts thus alleviating space and CPU requirements. The following function is provided to let the aligner know about such guiding constrains:

```
void  SetPattern(const vector<size_t>& pattern);
```

Pattern is a vector of hits specified by their zero-based coordinates as in the following example:

```
// the last parameter ommited to indicate nucl sequences
CNWAligner aligner (seq1, len1, seq2, len2);
// we want coordinates [99,119] and [129,159] on seq1 be aligned
// with [1099,1119] and [10099,10129] on seq2.
const size_t hits [] = { 99, 119, 1099, 1119, 129, 159, 10099, 10129 };
vector<size_t> pattern ( hits, hits + sizeof(hits)/sizeof(hits[0]) );
aligner.SetPattern(pattern);
```

Computing

To start computations, call ***Run()***, which returns the averall alignment score having aligned the sequences. Score is a scalar value associated with the alignment and depending on parameters of the alignment. The global alignment algorithms align two sequences so that the score is the maximal over all possible alignments.

Alignment transcript

The immediate output of the global alignment algorithms is a transcript. Transcript serves as a basic representation of alignments and is simply a string of elementary commands transforming the first sequence into the second one on a per-character basis. These commands (transcript characters) are (M)atch, (R)eplace, (I)nsert and (D)elete. For example, the alignment

```
TTC-ATCTCTAAATCTCTCTCATATATATCG
||| ||||||     |||| || ||| ||||
TTCGATCTCT-----TCTC-CAGATAAATCG
```

has a transcript

```
MMMIMMMMMMDDDDDMMMMDMMRMMMRMMMM
```

Several functions are available to retrieve and analyze the transcript:

```
// raw transcript
const vector<ETranscriptSymbol>* GetTranscript(void) const {
    return &m_Transcript;
}

// converted transcript vector
void GetTranscriptString(vector<char>* out) const;

// transcript parsers
size_t        GetLeftSeg(size_t* q0, size_t* q1,
                         size_t* s0, size_t* s1,
                         size_t min_size) const;
size_t        GetRightSeg(size_t* q0, size_t* q1,
                          size_t* s0, size_t* s1,
                          size_t min_size) const;
size_t        GetLongestSeg(size_t* q0, size_t* q1,
                            size_t* s0, size_t* s1) const;
```

The last three functions search for a continuous segment of matching characters and return it in sequence coordinates through `q0`, `q1`, `s0`, `s1`.

Alignment transcript is a simple yet complete representation of alignments that can be used to evaluate virtually every characteristic or detail of any particular alignment. Some of them, such as percent identity or the number of gaps or mismatches could be easily restored from the transcript alone, while others such as protein alignments' scores would require availability of original sequences.

Aligning sequences in linear space

CMMAligner is an interface to a linear space variant of the global alignment algorithm.This functionality is discussed in the following topics:

- The idea of the algorithm
- Implementation

The idea of the algorithm

That the classical global alignment algorithm requires quadratic space could be a serious restriction in sequence alignment. One way to deal with it is to use alignment patterns. Another approach was first introduced by Hirschberg and became known as a divide-and-conquer strategy. At a coarse level, it suggests computating of scores for partial alignments starting from two opposite corners of the dynamic programming matrix while keeping only those of them located at the middle rows or columns. Then after the analysis of the adjacent scores it is possible to determine cells on those lines through which the global alignment's backtrace path will go. This approach reduces space to linear while only doubling the worst-case time bound. For details see, for example, the Dan Gusfield's "Algorithms on Strings, Trees and Sequences".

Implementation

CMMAligner inherites its public interface from ***CNWAligner***. The only additional method allows to toggle multiple-threaded version of the algorithm.

The divide-and-conquer strategy suggests natural parrallelization where blocks of the dynamic programming matrix are evaluated simultaneously. A theoretical acceleration limit imposed by the current implementation is 0.5. In order to use multiple-threaded version, call ***EnableMultipleThreads()***. The number of simultaneously running threads will not exceed the number of CPUs installed on your system.

When comparing alignments produced with the linear-space version with those produced by ***CNWAligner***, be ready to find many of them similar though not exactly the same. This is normal, since several optimal alignmens may exist for each pair of sequences.

Computing spliced sequences alignments

This functionality is discussed in the following topics:

- The problem
- Implementation

The problem

The spliced sequence alignment arises as an attempt to address the problem of eukaryotic gene structure recognition. Tools based on spliced alignments expoit the idea of comparing genomic sequences to their transcribed and spliced products, such as mRna, cDna or EST sequences. The final objective for all spliced alignment algorithms is to come up with a combination of segments on the genomic sequence that:

- makes up a sequence very similar to the spliced product, when the segments are concatenated.
- satisfies certain statistically determined conditions, such as consensus splice sites and lengths of introns.

According to the classical eukaryotic transcription and splicing mechanism, pieces of genomic sequence do not get shuffled. Therefore, one way of revealing the original exons could be to align the spliced product with its parent gene globally. However, due to the specificity of the process in which the spliced product is constructed, the generic global alignment with the affine penalty model may not be enough. To address this accurately, dynamic programming recurrences should specifically account for introns and splices signals.

Algorithms described in this chapter exploit this idea and address a refined spliced alignment problem presuming that

- the genomic sequence contains only one location from which the spliced product could have originated.
- the spliced product and the genomic sequence are in plus strand.
- the Poly(A) tail and any other chunks of the product not created through the splicing were cut off, although a moderate level of sequencing errors on genomic, spliced, or both sequences is allowed.

In other words, the library classes provide a basic spliced alignment algorithms to be used in more sophisticated applications. One real-life application, Splign, can be found under demo cases for the library.

Implementation

There is a small hierarchy of three classes involved in spliced alignment facilitating a quality/ performance trade-off in case of distorted sequences:

- ***CSplicedAligner***-abstract base for spliced aligners.
- ***CSplicedAligner16***-accounts for the three conventional splices (GT/AG, GC/AG, AT/AC) and a generic splice; uses 2 bytes per backtrace matrix cell. Use this class with high-quality genomic sequences.
- ***CSplicedAligner32***-accounts for the three conventionales and splices that could be produced by damaging bases of any conventional; uses 4 bytes per backtrace matrix cell. Use this classes with distorted genomic sequences.

The abstract base class for spliced aligners, ***CNWSplicedAligner***, inherites an interface from its parent, ***CNWAligner***, adding support for two new parameters: intron penalty and minimal intron size (the default is 50).

All classes assume that the spliced sequence is the first of the two input sequences passed. By default, the classes do not penalize gaps at the ends of the spliced sequence. Default intron penalties are chosen so that the 16-bit version is able able to pick out short exons while the 32-bit version is generally more conservative.

As with the generic global alignment, the immediate output of the algorithms is the alignment transcript. For a sake of spliced alignments, the transcript's alphabet is augmented to accomodate introns as a special sequence editing operation.

Formatting computed alignments

This functionality is discussed in the following topics:

- Formatter object

Formatter object

CNWFormatter is a single place where all different alignments representations are created. The only argument to its constructor is the aligner object that actually was or will be used to align the sequences.

The alignment must be computed prior to formatting. If the formatter is unable to find the computed alignment in the aligner that was referenced to the constructor, an exception will be thrown.

To format the alignment as a ***CSeq_align*** structure, call

```
void AsSeqAlign(CSeq_align* output) const;
```

To format it as text, call

```
void AsText(string* output, ETextFormatType type, size_t line_width = 100) const;
```

Supported text formats and their ***ETextFormatType*** constants are:

- Type 1 (eFormatType1):

```
TTC-ATCTCTAAATCTCTCTCATATATATCG
TTCGATCTCT-----TCTC-CAGATAAATCG
                      ^   ^
```

- Type 2 (eFormatType2):

```
TTC-ATCTCTAAATCTCTCTCATATATATCG
||| ||||||     |||| || ||| ||||
TTCGATCTCT-----TCTC-CAGATAAATCG
```

- Gapped FastA (eFormatFastA):

```
>SEQ1
TTC-ATCTCTAAATCTCTCTCATATATATCG
>SEQ2
TTCGATCTCT-----TCTC-CAGATAAATCG
```

- Table of exons (eFormatExonTable) - spliced alignments only. The exons are listed from left to right in a tab-separated columns. The columns represent sequence IDs, alignment lengths, percent identity, coordinates on the query (spliced) and the subject sequences and a short annotation including splice signals.
- Extended table of exons (eFormatExonTableEx) - spliced alignments only. In addition to the nine columns, the full alignment transcript is listed for every exon.
- ASN.1 (eFormatASN)